Generalizing Leftvalues

Lecture #12

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

In the last lecture we optimized our arithmetic expression generation and stack allocation generation in order to reduce the overall size of compiled LLVM files:

```
(base) charlesaverill@pop-os:~/Desktop/ecco$ head examples/arith_test && wc -l examples/arith_test
int main() {
        print 10 * 5 + 10 - 8 - 6 - 2 + 6 + 7 - 9 * 8 + 7 * 0; // -15
        print 0 + 9 + 5 + 0 + 9 + 9 + 4 - 3 * 5; // 21
        print 5 + 8 + 6 * 0 + 8 + 4 - 3; // 22
        print 6 + 9 - 6 + 4 * 6 - 6 + 0 + 7 + 2; // 36
        print 1 * 10 + 2 * 6 - 1 - 5 * 2; // 11
        print 4 * 4 * 9 * 2 * 0 * 9 + 7 + 0 * 9 + 8 * 7; // 63
        print 9 * 4 - 2 - 4 + 10 - 2 + 7 + 10 - 1 + 0; // 54
        print 1 - 2 * 4 * 7 * 1 - 6 - 9 + 6; // -64
        print 1 * 10 + 3 * 1 * 8 - 0 - 6 - 0 + 8 * 7; // 84
501 examples/arith_test
(base) charlesaverill@pop-os:~/Desktop/ecco$ ./scripts run examples/arith_test -O0 --output noopt.ll && wc -l noopt.ll
-----RUN-----
19412 noopt.ll
(base) charlesaverill@pop-os:~/Desktop/ecco$ ./scripts run examples/arith_test -O1 --output oneopt.ll && wc -l oneopt.ll
-----RUN-----
524 oneopt.ll
```

What's Next?

I would really like to add arrays to our language. Unfortunately, there is some groundwork that we need to set up first:

1. Pointer offsets need to scale, e.g. (`int *x = {0, 1, 2}; x + 1;` $\Rightarrow$ `x + 1` should really be `x + 8`, because the programmer is most likely trying to grab the second element in the array, so the offset should be scaled to the width of an int)
2. Leftvalues need to become more dynamic so that we can perform array accesses and assignments (`int x = &y; *x = 5;`)
3. Function calls need to be finished up (not really, but I'd like to be able to call malloc and get rid of our print statement)

We'll start by making leftvalues more dynamic. Pointer offsets will be hard to test because LLVM won't guarantee that our integers will be next to each other in memory.

The Goal

We want to add support for the following test:

```
int main() {
    int a; int b; int c;
    a = b = c = 3;
    print a; print b; print c;

    int x; int *y;
    y = &x; *y = 5;
    print x;

    int **z; z = &y; **z = 19;
    print x;
}
```

Ideally, this update would also let us do something like `int **q; q = &y; *q = &a; *y = 7; print a;`, but a frustrating bug gets in the way that will be resolved when we add support for local variables.

The Plan

1. Restructure assignment statements by converting the ASSIGN token to an operator rather than a keyword
2. Add right associativity to the Pratt parser (assignment is right-associative)
3. Designate ASTNodes as either lvalues or rvalues
4. Update the code generator to support these new features

Assignment **operator**

We want the assignment token (=) to act as an **operator**, rather than a **keyword** like it has been up to this point.

That means that we can get rid of our "Lvalue Identifier" token, as assignments will now be able to be chained like any other operator. E.g. in `x = y = z = 3;`, all of x, y, and z are labeled as Identifier nodes, whereas previously in `x = 3; y = 3; z = 3;`, they would all be labeled as Leftvalue Identifier nodes.

Using assignment as an operator also allows us to generalize what appears on the left side of the symbol to pointer expressions. Now we will be able to assign to the **value** of what a pointer points to, e.g.

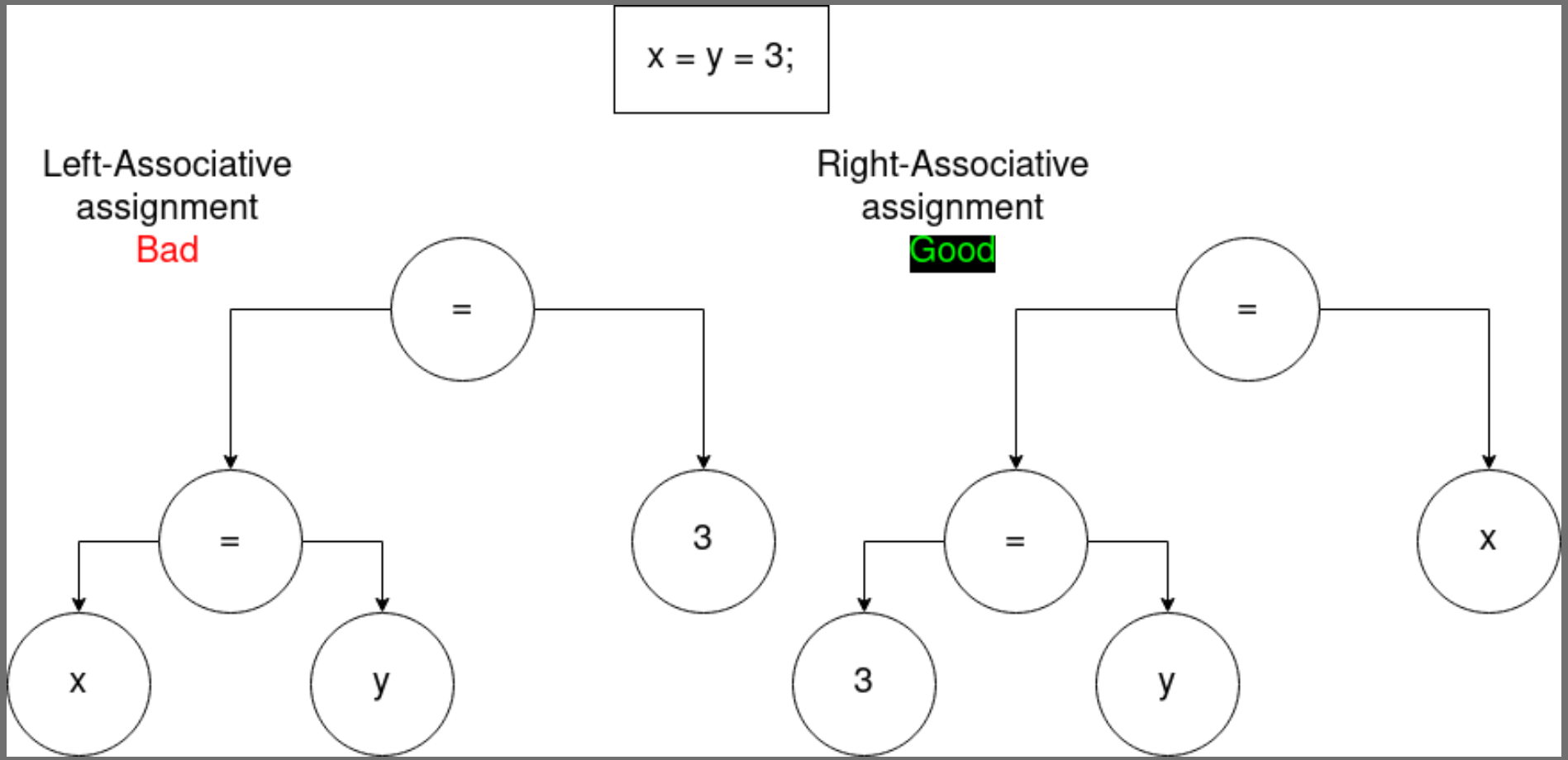
```
int x; int *y;
y = &x; *y = 5; print x; // 5
```

Assignment **operator**

Per our plan, I've added an `is_rvalue` field to the ASTNode class that defaults to False. Although most of the expressions we parse will be rvalues, we can't guarantee that something is an rvalue until we look at the next token in the stream, so we should assume every expression is an lvalue by default. (If we do the opposite, we have to do some extra tree traversal after expression parsing to un-rvalue lvalues and that's messy).

Next, we update our precedence table, and we add a list of right-associative operators.

Right Associativity



Right-Associative Pratt Parsing

Adding right associativity to our pratt parser is surprisingly easy. All we have to do is update our loop condition.

Previously, we just checked that our previous token's precedence was less than our current token's precedence. Now, we'll add a disjunctive case to check if an operator has the same precedence as the previous token, and is right-associative. Ta-da! Right associativity added.

Sort of. We need to update a few places to swap the left and right children and designate nodes as rvalues, but after that we've completed our parsing updates.

Code Generation Updates

There are three cases we need to update in `translate.py`:

- Identifiers - if it's an rvalue, load it like we did before, otherwise do nothing
- Assignments - if the right child is an Identifier, store it like we used to do in the Leftvalue Identifier case. If the right child is a dereference, call our new dereference store function
- Dereferences - if it's an rvalue, dereference like we did before, otherwise do nothing